Lightweight Asynchrony Using Parasitic Threads

KC Sivaramakrishnan Lukasz Ziarek Raghavendra Prasad Suresh Jagannathan

Purdue University

{chandras, lziarek, prasadr, suresh}@cs.purdue.edu

Abstract

Message-passing is an attractive thread coordination mechanism because it cleanly delineates points in an execution when threads communicate, and unifies synchronization and communication: a sender is allowed to proceed only when a receiver willing to accept the data being sent is available and vice versa. To enable greater performance, however, asynchronous or non-blocking extensions are usually provided that allow senders and receivers to proceed even if a matching partner is unavailable. Lightweight threads with synchronous message-passing can be used to encapsulate asynchronous message-passing operations, although such implementations have greater thread management costs that can negatively impact scalability and performance.

This paper introduces *parasitic threads*, a novel mechanism for expressing asynchronous computation, that combines the efficiency of a non-declarative solution with the ease of use provided by languages with first-class channels and lightweight threads. A *parasitic thread* is a lightweight data structure that encapsulates an asynchronous computation using the resources provided by a host thread. Parasitic threads need not execute cooperatively, impose no restrictions on the computations they encapsulate, or the communication actions they perform, and impose no additional burden on thread scheduling mechanisms.

We describe an implementation of parasitic threads in MLton, a whole-program optimizing compiler and runtime for Standard ML. Benchmark results indicate parasitic threads enable construction of scalable and efficient message-passing parallel programs.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Concurrent programming structures; D.1.3 [*Concurrent Programming*]: Parallel programming; D.3.1 [*Formal Definitions and Theory*]: Semantics

General Terms Design, Experimentation, Languages, Measurement, Performance

Keywords Lightweight threading, Message passing, Asynchronous communication, MLton

1. Introduction

Many parallel programs are constructed using message passing primitives to coordinate communication and synchronization activities among a collection of concurrently executing threads. Message passing explicitly delineates which data is transmitted from one thread to another, and identifies the points where communication takes place. Message passing systems also explicitly define synchronization points through synchronous data exchange primitives. These properties alleviate the need to reason about data-races, simplify program structure, and reduce algorithmic complexity.

Despite the semantic simplicity of synchronous message passing, performance requirements often entail the need for non-blocking or asynchronous communication. By allowing communication to overlap with computation, overall throughput can be increased. However, the performance gains afforded by performing communication asynchronously comes at a price. If the semantics of communication primitives is expressed in terms of non-declarative language with explicit buffer management and point to point communication between processors, correctly using non-blocking variants requires care to ensure previous operations on send or receive buffers have completed before the buffers are reused. Because communication is defined with respect to how buffered data is transmitted from one processor to another, messages must provide appropriate metadata to allow receivers to distinguish between various message types and senders. Thus, implementations of asynchronous communication defined in terms of buffer movement requires communicating threads to be aware of each other's internal semantics, limiting abstraction and composability. To illustrate, consider the following code fragment:

```
1  void worker (master, workSize) {
2      int tag = 0;
3      for (int i=0; i < workSize; i++) {
4          tag = 0; /* Data request */
5          send (buf, master, tag);
6          recv (buf, master);
7          Data* data = parse (buf);
8          HashBucketID b = classify (data);
9          tag = 1; /* Update */
10         writeBuf (buf, b);
11         send (buf, master, tag);
12     }
13 }
```

This code fragment outlines the worker's side of a concurrent program to plot a histogram. The master partitions an input dataset into chunks to distribute among its workers. Each worker sends a request to the master for the next data element (lines 5-6), classifies the data (line 8), and sends back an identifier indicating the bucket to which the element belongs (line 11); the histogram is then updated by the master. Due to the absence of dynamic channel creation, the request for the next data element and update has to be multiplexed on the same static channel, with explicit tags. The unavailability of typed channels places the onus on the programmer to parse the byte buffer.

DAMP'10, January 19, 2010, Madrid, Spain.

Without dynamic channel creation and lightweight threading, the opportunities for exploiting fine grained concurrency is difficult. Although there exists an opportunity to send results to the master in the current iteration (line 11) concurrently with the request of the data element in the next iteration (line 5), there is no convenient means of expressing this fact. Simply using a non-blocking send call on line 11 would not help. While that would allow the write of the data to the send buffer to proceed concurrently with initiating the next iteration, it would not allow the send to take place concurrently with the next request since both send operations are directed to the same receiver.

In contrast to defining communication semantics in terms of data movement among buffers, languages such as Erlang (3), Haskell (8; 7), Concurrent ML (21; 20), and F# (24) allow threads to communicate via first-class channels. This allows encoding of abstract communication protocols and alleviates explicit message management. For instance, a private channel can be allocated between two communicating threads to be used as a unique vessel for communication. Combined with lightweight threads, these languages can express asynchronous computation by creating a new thread of control that communicates over shared channels.

```
1  fun doWork (master, workSize) =
2   let fun worker () =
3         let val data = recv (master)
4             val hashBucket = classify (data)
5         in send (hashBucket, ())
6         end
7         fun loop (workSize) =
8           if workSize = 0 then ()
9           else (spawn (worker);
10                loop(workSize-1))
11  in loop(workSize)
12  end
13
14 fun categoryServer (ch, c) =
15  (recv (ch); c := !c + 1;
16   categoryServer (ch, c))
```

Consider the non-declarative program shown earlier expressed above in Concurrent ML, a dialect of ML that defines synchronous first-class communication channels. Here, the master spawns a server for each category (a `categoryServer`) in the histogram and associates each with a unique channel. The `classify` function categorizes the data and returns a channel corresponding to the category. Workers can now directly communicate with these servers to update the bucket, instead of communicating with the master. The servers waits for a communication from a worker, and increments a counter that holds the value of their category.

This solution is more abstract than the first, and potentially exhibits greater concurrency: updates to different categories by different workers can happen simultaneously without the need for a centralized master to mediate and parse message data. Unfortunately, the solution also places a large burden on the language runtime because the number of threads created is a function of the categories desired in the histogram. While a language runtime can be configured to use thread pools to limit the number of underlying threads, such a solution does not lead to improved scalability. This occurs because each category server repeatedly blocks waiting for a response from a worker; depending upon the number of categories used, the number of available threads in a thread pool can be easily exhausted.

In this paper, we investigate an application-transparent technique for creating and managing asynchronous computation which retains the high-level abstraction afforded by first-class channels and lightweight threads for expressing asynchrony, *without* imposing additional thread management and scheduling overheads. To do this, we introduce a new construct called a *parasitic thread*. The expression: `parasite(e)` creates a new parasitic thread to evaluate expression `e`. Unlike a regular thread, a parasitic thread executes using the execution context of the parent which creates the parasite. The parent, or host, thread can support an arbitrary number of parasites. When a parasite is preempted, another parasite on the host can be chosen for execution: switching between parasites is a thread-local operation, and in the common case, entails no stack movement. This results in a zero-copy context switch with low execution overheads.

A parasite no longer becomes schedulable on a host when it performs a blocking action (e.g., a synchronous communication operation, or I/O). Such parasites can resume execution once the conditions that caused it block no longer hold. Thus, parasitic threads are not scheduled using the language runtime; instead they self-schedule in a demand-driven style based on flow properties dictated by the communication and I/O actions they perform. For example, consider a parasite that performs a communication action such as a synchronous send on a channel c that would cause it block. When a matching operation occurs on c, the parasite is notified that it can resume execution. Notably, there is no locking required to effect this state change, and no involvement from a global thread scheduler. Thus, a thread stack now contains a number of continuations, one for each parasitic thread. A thread's metadata structure is augmented to allow indexing into the stack to access any housed parasitic thread.

In the above example, each category server would be implemented as a parasite, and each worker would be implemented as a regular lightweight thread. Category servers are allocated on the master thread, but become blocked once they perform a synchronous receive operation that is waiting for data from a worker. When a worker performs a matching send because it classifies its current data as belonging to that category, the category server parasite resumes execution, increments its counter, and loops. The parasite can continue to execute provided there are additional communication requests from other workers available.

The remainder of this paper describes the operational behavior of parasitic threads, and their implementation in MLton, a whole-program optimizing compiler for Standard ML. Section 2 describes the overall system design, and presents a formal operational semantics. Implementation details are provided in Section 3. Benchmark results, presented in Section 4, on both communication-intensive micro-benchmarks and well-studied (unbalanced) parallel benchmarks validate the scalability and efficiency of our technique.

2. System Design and Semantics

In this section, we formalize our system design. Our system supports two types of threads: hosts and parasites. Host threads map directly to the threading model provided by the language runtime, whereas parasitic threads are implemented as raw frames living within the stack space of a given host thread. A host thread can hold an arbitrary number of parasitic threads. In this sense, a parasitic thread views its host in much the same way as a user-level thread might view a kernel-level one that it executes on. The key characteristic that distinguishes parasitic threads from lightweight ones is that blocking synchronization actions performed by a parasitic thread does not cause the host to block in turn. Because parasites do not require cooperative scheduling, and can be preempted, there is substantial flexibility to avoid deadlocks, ensure fairness, etc. regardless of the hosts they are allocated on.

In the following figures, host threads are depicted as rounded rectangles, parasitic threads are represented as blocks within their hosts, and each processor as a queue of host threads. The parasite which is currently executing on a given host and its stack is represented as a block with solid edges; suspended parasites and their stacks are represented as blocks with dotted edges.

Host threads can be viewed as a collection of parasitic threads all executing within the same stack space. When a thread is initially created it contains one such parasitic computation, namely the expression it was given to evaluate when it was spawned. Threads and parasites communicate with one another via synchronous message passing. Communication actions on the same channel are paired non-deterministically.

Our system supports synchronous message passing over channels as well as user-defined blocking events. During synchronous communication, the parasite which initiates the synchronous communication *blocks* (either sending or receiving on a channel), if a matching communication is not already available. It is subsequently unblocked by the parasite that terminates the protocol by performing the opposite communication action (i.e. a matching send or receive). Similarly, a parasite may block on an event (such as I/O). This parasite is available to execute once the event is triggered. Once the conditions that prevent continued execution of the parasite becomes resolved, the parasite enters a suspended state, and can be resumed on the host.

Fig. 1 shows the steps involved in such a communication, or blocking event. Initially, the parasite S_1 performs a blocking action on a channel or event, abstractly depicted as a circle. Hence, S_1 blocks. Part 2 of the figure shows that thread T_1 which hosts S_1 continues execution by switching to a suspended (runnable) parasite S_m . Notice that although S_1 is blocked, the thread hosting the parasite is not. S_1 becomes unblocked by a synchronizing send on the same channel. Part 3 of the figure shows parasite S_1 on processor P_n invoking a send action on the channel. Since a blocked receiver is already present on the channel, sender S_1 does not block. The value is sent to the blocked parasite which unblocks the parasite. We envision parasites to be communication intensive, spending most of their time in the blocked state waiting for a synchronization action. When unblocked, parasites are implicitly rescheduled for re execution on their host.

2.1 Formal Semantics

We define our system formally through the use of a formal operational semantics and model host threads in terms of sets of stacks and parasites as stacks. Transitions in our formalism are defined through stack based operations. Our semantics is defined in terms of a core call-by-value functional language with threading and communication primitives. New threads of control, or host threads, are explicitly created through a `spawn` primitive. To model parasitic thread creation we extend this simple language with a primitive `parasite`. Therefore, computation in our system is split between host threads, which behave as typical threads in a language runtime, and parasitic threads, which behave as asynchronous operations with regard to their host.

We formalize the behavior of parasitic threads in terms of an operational semantics expressed using a CEK machine (14). A typical CEK machine is small-step operational definition that operates over program states. A state is composed of an expression being evaluated, an environment, and the continuation (the remainder of the computation) of the expression. The continuation is modeled as a *stack* of frames.

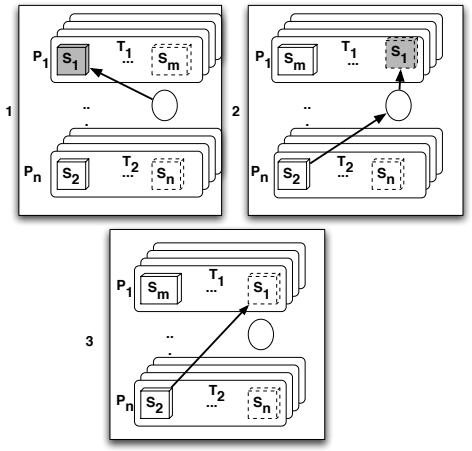

Figure 1. The block and unblocking of parasitic threads via synchronous communication.

2.1.1 Language

In the following, we write $\bar{\mathtt{k}}$ to denote a set of zero or more elements and $\emptyset$ as the empty set. We write $x : l$ to mean the concatenation of x to a sequence l where . denotes an empty sequence. Evaluation rules are applied up to commutativity of parallel composition ($\|$). Relevant domains and meta-variables used in the semantics are shown in Fig. 2.

In our semantics, we use stack frames to capture intermediate computation state, to store environment bindings, to block computations waiting for synchronization, and to define the order of evaluation. We define eight unique types of frames: return frames, argument frames, function frames, receive frames, send frames, send value frames, and receive and send blocked frames. The return frame pushes the resulting value from evaluating an expression on to the top of the stack. The value pushed on top of the stack gets propagated to the frame beneath the return frame (see Fig. 3 `Return Transitions`). The receive and send blocked frames signify that a thread is blocked on a send or receive on a global channel. They are pushed on top of the stack to prevent further evaluation of the given parasitic computation. Only a communication across a global channel can pop a blocked frame. Once this occurs, the parasitic thread can resume its execution. Argument and function frames enforce left-to-right order of evaluation. order. Similarly, the send and send value frames define the left to right evaluation of the `send` primitive.

Our semantics is defined with respect to a global mapping ($\mathcal{T}$) from thread identifiers (`t`) to thread states (`s`). A thread is a pair composed of a thread identifier and a thread state. A thread state (`s`) is a CEK machine state extended with support for parasitic threads. Therefore, a thread is a collection of stacks, one for each parasitic thread. A concrete thread state can be in one of three configurations: a control state, a return state, or a halt state. A *control state* is composed of an expression (`e`), the current environment (`r`) - a mapping between variables and values, the current stack (`k`), as well as a set of parasitic computations ($\bar{\mathtt{k}}$). A *return state* is simply a collection of parasitic computations ($\bar{\mathtt{k}}$). The *halt state* is reached when all parasitic threads in a given thread have completed. A thread, therefore, is composed of a collection of parasitic threads executing within its stack space. When a thread transitions to a control state, one of the thread's parasites is chosen to be evaluated.

$$
\begin{array}{rcl}
\mathtt{e} \in \mathit{Exp} & ::= & \mathtt{x} \mid \mathtt{v} \mid \mathtt{e(e)} \mid \mathtt{spawn(e)} \mid \mathtt{parasite(e)} \\
& \mid & \mathtt{Chan()} \mid \mathtt{send(e,e)} \mid \mathtt{recv(e)} \\
\mathtt{v} \in \mathit{Val} & ::= & \mathtt{unit} \mid (\lambda \mathtt{x.e,r}) \mid \mathtt{c} \mid \kappa
\end{array}
$$

$$
\begin{array}{rcrcl}
\kappa & \in & \mathit{Constant} & & \\
\mathtt{c} & \in & \mathit{Channel} & & \\
\mathtt{x} & \in & \mathit{Var} & & \\
\mathtt{t} & \in & \mathit{ThreadID} & & \\
\mathtt{r} & \in & \mathit{Env} & = & \mathit{Var} \rightarrow \mathit{Value} \\
\mathtt{k} & \in & \mathit{Cont} & = & \mathit{Frame}* \\
\mathtt{v} & \in & \mathit{Value} & = & \mathit{Unit} + \mathit{Closure} + \mathit{Channel} + \mathit{Constant} \\
(\lambda \mathtt{x.e,r}) & \in & \mathit{Closure} & = & \mathit{LambdaExp} \times \mathit{Env}
\end{array}
$$

$$
\begin{array}{rcrcl}
\mathit{ret}\lceil \mathtt{v} \rceil & \in & \mathit{RetFrame} & = & \mathit{Value} \\
\mathit{arg}\lceil \mathtt{e,r} \rceil & \in & \mathit{ArgFrame} & = & \mathit{Exp} \times \mathit{Env} \\
\mathit{fun}\lceil \mathtt{v} \rceil & \in & \mathit{FunFrame} & = & \mathit{Value} \\
\mathit{recv}\lceil \rceil & \in & \mathit{RecvFrame} & = & \mathit{Empty} \\
\mathit{sval}\lceil \mathtt{e,r} \rceil & \in & \mathit{SValFrame} & = & \mathit{Exp} \times \mathit{Env} \\
\mathit{send}\lceil \mathtt{v} \rceil & \in & \mathit{SendFrame} & = & \mathit{Value} \\
\mathit{rblock}\lceil \mathtt{v} \rceil & \in & \mathit{RBlockFrame} & = & \mathit{Value} \\
\mathit{sblock}\lceil \mathtt{v}_1, \mathtt{v}_2 \rceil & \in & \mathit{SBlockFrame} & = & \mathit{Channel} \times \mathit{Value}
\end{array}
$$

$$
\begin{array}{rcrcl}
\mathcal{T} & \in & \mathit{GlobalMap} & = & \mathit{ThreadID} \rightarrow \mathit{ThreadState} \\
\mathtt{s} & \in & \mathit{ThreadState} & = & \mathit{ControlState} + \mathit{ReturnState} + \mathit{HaltState} \\
\langle \mathtt{e,r,k,\bar{k}} \rangle & \in & \mathit{ControlState} & = & \mathit{Exp} \times \mathit{Env} \times \mathit{Cont} \times \mathit{Cont}* \\
\langle \mathtt{\bar{k}} \rangle & \in & \mathit{ReturnState} & = & \mathit{Cont}* \\
\mathtt{halt(v)} & \in & \mathit{HaltState} & = & \mathit{Value}
\end{array}
$$

Figure 2. Domains for the CEK machines extended with threads and parasites.

A thread switches evaluation between its various parasitic threads non-deterministically when it transitions to a return state.

2.1.2 CEK Machine Semantics

The rules given in Fig. 3 and Fig. 4 define the transitions of the CEK machine. There are three types of transitions: control transitions, return transitions, and global transitions. Control and return transitions are thread local actions, while global transitions affect global state. We utilize the two types of local transitions to distinguish between states in which an expression is being evaluated from those in which an expression has already been evaluated to a value. In the latter case, the value is propagated to its continuation. Global transitions are transitions which require global coordination, such as the creation of a new channel or thread, or a communication action.

There are four rules which define global transitions given in Fig. 4. Rule (`Local Evaluation`) states that a thread with thread state $\mathtt{s}$ can transition to a new state $\mathtt{s}'$ if it can take a local transition from $\mathtt{s}$ to $\mathtt{s}'$. This rule subsumes thread and parasite scheduling, and defines global state change in terms of operations performed by individual threads. The second rule, (`Channel`), defines the creation of a new global channel. The (`Spawn`) rule governs the creation of a new thread; this rule generates a unique thread identifier and begins the evaluation of the spawned expression (`e`) in the parent thread's (`t`) environment (`r`).

The last global rule, (`Communication`), deals with communication across a shared channel (`c`) by two threads. The rule finds two threads, one blocked sending on the channel and the other blocked receiving. The blocked frames are popped off of each thread's stack and return frames are pushed in their place. The sender will contain a return frame with the unit value and the receiver will contain a return frame with the value transmitted across the channel (see step 3 of Fig. 1).

There are seven rules which define local control transitions. Because the definition of these rules are standard, we omit their explanation here, with the exception of the last rule (`Parasite`). This rule models the creation of a new parasitic thread within the current thread. The currently evaluating parasitic thread is added back to the set of parasites with a unit return value pushed on its stack. The expression is evaluated in a new parasitic thread constructed with the environment of the parent and an empty stack. Thread execution undertakes evaluation of the expression associated with this new parasite.

There are eight rules which define local return transitions. These rules, like local control transitions, are mostly standard. We comment on the three rules that involve thread and parasite management. Rule (`Halt Thread`) defines thread termination via a transition to a halt state. A thread transitions to a halt state if it has no active parasites and its stack is empty except for a return frame. Parasites themselves are removed by the (`Parasite Halt`) rule. The return value of a thread is thus defined as the last value produce by its last parasite. The lifetime of a thread is bounded by the parasites which inhabit it. Rule (`Local Communication`) is similar to the global communication rule, but is defined as a local transition when both communicating parties reside in the same thread. The transition pops off both blocked frames for the communicating parasitic threads. It also pushes new return frames, the value being sent on to the receiver's stack and the unit value on top of the sender's stack.

2.1.3 Safety

Consider a replacement function $\mathcal{R}$ which transforms a program containing `parasite(e)` expressions into an equivalent program in which these expressions are replaced by `spawn(e)` expressions. We can now define a *Safety* theorem that states the equivalence between these two programs.

Theorem[Safety] If
$\mathcal{T}[\mathtt{t}_0 \mapsto \langle \mathtt{e}_0, \mathtt{r}, ., \emptyset \rangle] \rightarrow^* \mathcal{T}[\mathtt{t}_0 \mapsto \mathtt{halt}(\mathtt{v}_0) ... \mathtt{t}_n \mapsto \mathtt{halt}(\mathtt{v}_n)]$ then there exists an evaluation sequence
$\mathcal{T}[\mathtt{t}_0 \mapsto \langle \mathcal{R}(\mathtt{e}_0), \mathtt{r}, ., \emptyset \rangle] \rightarrow^* \mathcal{T}[\mathtt{t}_0 \mapsto \mathtt{halt}(\mathtt{v}'_0) ... \mathtt{t}_n \mapsto \mathtt{halt}(\mathtt{v}'_n) ... \mathtt{t}_m \mapsto \mathtt{halt}(\mathtt{v}'_m)]$ where $m \leq n$ and $v_0 \cong v'_0 ... v_n \cong v'_n$. ($v \cong v'$ defines standard value equivalence up to alpha-renaming of identifiers for channels and bound variables.)

Note that the transformed program can have more threads than the original since the (`ParasiteHalt`) rule removes halted parasites from their host threads.

3. System Implementation

We have implemented our system in Multi-MLton,a parallel extension of MLton (16), a whole-program optimizing compiler for

Control Transitions

$$
\begin{array}{lrcl}
(\texttt{Constant}) & \langle \kappa, r, k, \bar{k}\rangle & \longrightarrow & \langle \mathit{ret}\lceil \kappa \rceil : k \| \bar{k}\rangle \\
(\texttt{Variable}) & \langle x, r, k, \bar{k}\rangle & \longrightarrow & \langle \mathit{ret}\lceil r(x) \rceil : k \| \bar{k}\rangle \\
(\texttt{Closure}) & \langle \lambda x.e, r, k, \bar{k}\rangle & \longrightarrow & \langle \mathit{ret}\lceil \lambda x.e, r \rceil : k \| \bar{k}\rangle \\
(\texttt{Application}) & \langle (e_1 e_2), r, k, \bar{k}\rangle & \longrightarrow & \langle e_1, r, \mathit{arg}\lceil e_2, r \rceil : k, \bar{k}\rangle \\
(\texttt{Send}) & \langle \texttt{send}(e_1, e_2), r, k, \bar{k}\rangle & \longrightarrow & \langle e_1, r, \mathit{sval}\lceil e_2, r \rceil : k, \bar{k}\rangle \\
(\texttt{Receive}) & \langle \texttt{recv}(e), r, k, \bar{k}\rangle & \longrightarrow & \langle e, r, \mathit{recv}\lceil \rceil : k, \bar{k}\rangle \\
(\texttt{Parasite}) & \langle \texttt{parasite}(e), r, k, \bar{k}\rangle & \longrightarrow & \langle e, r, ., (\mathit{ret}\lceil \texttt{unit} \rceil : k) \| \bar{k}\rangle
\end{array}
$$

Return Transitions

$$
\begin{array}{lrcl}
(\texttt{ThreadHalt}) & \langle \mathit{ret}\lceil v \rceil : . \| \emptyset\rangle & \longrightarrow & \texttt{halt}(v) \\
(\texttt{ParasiteHalt}) & \langle \mathit{ret}\lceil v \rceil : . \| \bar{k}\rangle & \longrightarrow & \langle \bar{k}\rangle \\
(\texttt{Argument}) & \langle \mathit{ret}\lceil v \rceil : \mathit{arg}\lceil e, r \rceil : k \| \bar{k}\rangle & \longrightarrow & \langle e, r, \mathit{fun}\lceil v \rceil : k, \bar{k}\rangle \\
(\texttt{Function}) & \langle \mathit{ret}\lceil v \rceil : \mathit{fun}\lceil \lambda x.e, r \rceil : k \| \bar{k}\rangle & \longrightarrow & \langle e, r[x \mapsto v], k, \bar{k}\rangle \\
(\texttt{SendValue}) & \langle \mathit{ret}\lceil c \rceil : \mathit{sval}\lceil e, r \rceil : k \| \bar{k}\rangle & \longrightarrow & \langle e, r, \mathit{send}\lceil c \rceil : k, \bar{k}\rangle \\
(\texttt{SendBlock}) & \langle \mathit{ret}\lceil v \rceil : \mathit{send}\lceil c \rceil : k \| \bar{k}\rangle & \longrightarrow & \langle \mathit{sblock}\lceil c, v \rceil : k \| \bar{k}\rangle \\
(\texttt{ReceiveBlock}) & \langle \mathit{ret}\lceil c \rceil : \mathit{recv}\lceil \rceil : k \| \bar{k}\rangle & \longrightarrow & \langle \mathit{rblock}\lceil c \rceil : k \| \bar{k}\rangle \\
(\texttt{LocalCommunication}) & \langle (\mathit{rblock}\lceil c \rceil : k_1) \| (\mathit{sblock}\lceil c, v \rceil : k_2) \| \bar{k}\rangle & \longrightarrow & \langle (\mathit{ret}\lceil v \rceil : k_1) \| (\mathit{ret}\lceil \texttt{unit} \rceil : k_2) \| \bar{k}\rangle
\end{array}
$$

Figure 3. Local evaluation defining both control and return transitions.

Global Transitions

$$
\begin{array}{ll}
(\texttt{LocalEvaluation}) & \dfrac{s \longrightarrow s'}{\langle \mathcal{T}[t \mapsto s]\rangle \longrightarrow \langle \mathcal{T}[t \mapsto s']\rangle} \\[2ex]
(\texttt{Channel}) & \dfrac{c \; \mathit{fresh}}{\langle \mathcal{T}[t \mapsto \langle \texttt{Chan}(), r, k, \bar{k}\rangle]\rangle \longrightarrow \langle \mathcal{T}[t \mapsto \langle \mathit{ret}\lceil c \rceil : k \| \bar{k}\rangle]\rangle} \\[2ex]
(\texttt{Spawn}) & \dfrac{t' \; \mathit{fresh}}{\langle \mathcal{T}[t \mapsto \langle \texttt{spawn}(e), r, k, \bar{k}\rangle]\rangle \longrightarrow \langle \mathcal{T}[t \mapsto \langle \mathit{ret}\lceil \texttt{unit} \rceil : k \| \bar{k}\rangle, t' \mapsto \langle e, r, ., \emptyset\rangle]\rangle} \\[2ex]
(\texttt{Communication}) & \dfrac{\begin{array}{c} s_1 = \langle \mathit{rblock}\lceil c \rceil : k_1 \| \bar{k_1}\rangle, s_2 = \langle \mathit{sblock}\lceil c, v \rceil : k_2 \| \bar{k_2}\rangle \\ s_1' = \langle \mathit{ret}\lceil v \rceil : k_1 \| \bar{k_1}\rangle, s_2' = \langle \mathit{ret}\lceil \texttt{unit} \rceil : k_2 \| \bar{k_2}\rangle \end{array}}{\langle \mathcal{T}[t_1 \mapsto s_1 t_2 \mapsto s_2]\rangle \longrightarrow \langle \mathcal{T}[t_1 \mapsto s_1' t_2 \mapsto s_2']\rangle}
\end{array}
$$

Figure 4. Global evaluation rules defined in terms of thread states ($\mathcal{T}$).

Standard ML (SML) (15). MLton compiles ML programs to both native code as well as C; the results described in this paper are based on code compiled to C and then passed to gcc version 4.1.2. Code compiled with MLton exhibits excellent runtime performance, often on par, if not superior to, optimized implementations of other managed languages like Java or Haskell (7).

Multi-MLton extends MLton with multi-core support, library primitives for efficient lightweight thread creation and management, as well as CML (21), an optimized synchronous message passing extension to SML. In this section, we present the runtime details of parasitic threads and their implementation within MLton.

3.1 Host threads

Our implementation is targeted at high performance SMP platforms. The implementation supports *m* host threads running on top of *n* processors, where $m \geq n$. Each processor runs a single Posix thread which maintains a queue of host threads. Each thread has a contiguous stack, allocated on the MLton heap, which can dynamically grow and shrink during execution. The information associated with the currently executing host thread is cached in the runtime state associated with each processor to improve performance. On a context switch, this information is written back to the host thread. The code to accomplish the context switch is generated by the compiler and is highly optimized.

A new host thread, created using `spawn`, is placed in a processor queue, implemented as a lock-free data structure, in a round-robin fashion. When there are no host threads in a processor queue, the pthread is suspended, to be woken up when a new host thread is added.

3.2 Parasitic threads

Unlike host threads, parasitic threads are implemented as raw stack frames. The expression `parasite(e)` pushes a new frame to evaluate expression `e`, similar to a function call. We capture the stack top at the point of invocation. This corresponds to the caller's continuation and is a *de facto* boundary between the parasite and its host (or potentially another parasite). If the parasite is not blocked or preempted, the computation runs to completion and control returns to the caller, just as if the caller made a non-tail procedure call. In our implementation, parasite scheduling is round-robin, with the parasite at the bottom of the host thread the next to run, but there is no restriction on the specific local scheduling policy adopted by a host thread.

A stack of parasites occupy each host thread. At any given point, there is one running parasite, and a number of suspended and blocked parasites on a host. The running parasite need not be at the top of the host thread stack, as we explain below. When a new host thread is created to evaluate an expression, a new parasite is generated to evaluate the expression. When the parasite bound to a host completes and there are no other parasite executing on the host thread, the host thread is reclaimed.

3.3 Compiler support

As mentioned above, when a parasite blocks or preempted, we resume the parasite at the bottom of the stack. Since this parasite has other parasites surrounding it, care must be taken to ensure that it does not overwrite context information associated with these other parasites; this can happen if it pushes a new stack frame on a non-tail call or requires additional space for temporaries that cause its local context to overflow onto the context of some other active parasite. We utilize the facilities provided by the MLton compiler to implement highly optimized context switching among the parasites that takes these issues into account.

MLton emits information associated with every stack frame including the size of frames, the location of pointers, etc. This information is utilized by the garbage collector when it walks the host thread stack during garbage collection. Hence, for every stack frame we statically know the maximum size it can grow during execution. We leverage this information to overcome the problem of overwriting parasites when a suspended or blocked parasite resumes execution.

When a parasite is created, we insert a new frame associated with the computation it encapsulates with a maximum frame size offset from the bottom of the current frame; this frame size is determined by the compiler, and ensures sufficient space for all temporaries allocated by the procedure associated with the frame. In addition, each parasite has extra stack space allocated to it for non-tail calls it may perform; such calls requires a new frame to be pushed onto the stack. Compiler analysis of the frame's continuation provides a handle on possible non-tail calls the parasite may execute, and the storage requirements of these calls. The extra stack space allocated to a parasite provides a coarse approximation to the actual storage required, up to a predefined limit. When the parasite inserts a new frame that would cause an overflow of available stack space, its frames are copied to the top of the stack. If the new frame fits into existing space allocated for it, it is inserted at the current position. We can find out if a frame fits in available space since the compiler provides the maximum size for this frame. It is possible even with overflow to not have to copy the parasite. This is because stack storage above the parasite may be free because the parasite that previously occupied that space has been moved; in this case, that space can be immediately reused if needed. MLton also employs aggressive inlining and loop unrolling optimizations on computations known to be encapsulated within a parasite to reduce additional stack growth. Of course, parasites executing on top of the stack can insert new frames with no additional overhead.

3.4 Thread Metadata

A host thread's metadata is augmented with information about the parasites it holds. We maintain a linked list of parasites with the head of the list pointing to the parasite on the bottom of the host thread's stack and the tail to the top parasite. Hence, resuming the parasite at the bottom of a host thread's stack entails a single pointer indirection. If the parasite at the bottom runs to completion, it resumes the parasite immediately on top of it, traversing the linked list. When a parasite is moved to the top of the stack, we update the host's metadata to reflect the change.

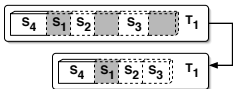

Figure 5. Stack with holes and the result of stack compaction

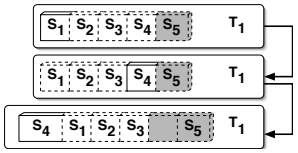

Figure 6. Preemption and parasite context switch

Each parasite also maintains a small amount of state information. Specifically, a parasite maintains a flag indicating whether it is runnable or blocked waiting for a synchronization action. When a blocked parasite is encountered during traversal, we skip it and continue to execute the next suspended parasite. Thus, a host thread's stack may be interspersed with holes as shown in Fig. 5 that arise when parasites requiring more storage than available at their current location in the host thread's stack move to the top of the stack. The figure shows parasite S_4 running on top of the stack; S_2 and S_3 are in suspended states. Moreover, S_1 is in a blocked state waiting for a synchronization action. Holes are represented as shaded blocks with no labels. Holes are not repaired as they occur since that may entail a full stack copy. Instead, they are fixed during garbage collection, or when the stack is grown or shrunk. The latter operation is performed by the MLton runtime when the stack requirements of the host thread exceed its initial allocation. When a stack is compacted, we walk the host thread's metadata and copy the parasites to the new location. Figure 5 shows the result of compaction where the stack has no holes.

3.5 Preemption

A parasitic thread represents asynchronous computation from the perspective of the host which created it. We impose no restriction on the kinds of computations encapsulated within a parasitic thread; indeed, as described above, `spawn`ed computations are themselves represented as parasites. Cooperative scheduling would enable parasites to share a host's resources, but with increased program complexity, and loss of fairness guarantees due to deadlock or livelock. To avoid these issues, parasites are scheduled preemptively on a host.

To preempt a parasite, the stack top register is updated to point to the top of the next parasite waiting to be run. We leverage compiler support to restore other registers. Thus, the preempted parasite becomes suspended. Notably, this context switch operation does not involve any copying. If the newly scheduled parasite performs a non-tail call that causes its execution context to overflow its allocated region on the host thread's stack, the parasite is copied to the top of the host thread stack; the hole it leaves behind is compacted as described earlier.

Fig. 6 shows the steps involved in preempting a parasite. Initially, the figure shows parasite S_1 is running with suspended parasites below it on the stack. On a timer interrupt, S_1 is suspended and parasite S_4 is resumed, skipping the blocked parasite S_5 . As long as S_4 does not insert a new frame, it continues to run. If S_4 grows and moves to top, it leaves in its wake a hole, which is eventually compacted.

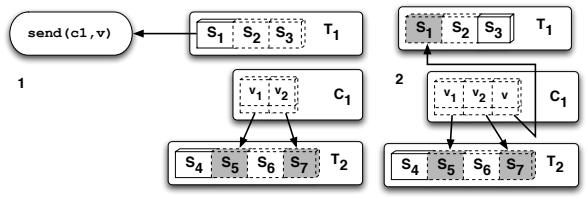

Figure 7. Parasite performing a send action on a channel, which blocks

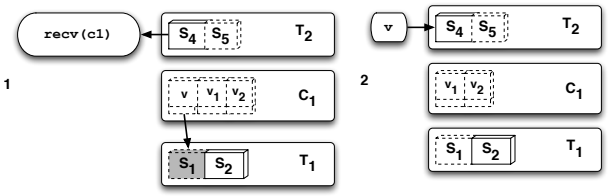

Figure 8. Parasite performing a matching receive on a channel

3.6 Synchronous communication

Parasites interact by sending messages synchronously to each other over typed channels. A synchronous communication is satisfied only when it pairs up with a matching communication action. If there is no matching communication available on the channel, the parasite becomes blocked, and resumes only once a matching communication becomes available.

Fig. 7 shows the effect of performing a `send` on a channel with no blocked receivers. The channel C_1 has a queue of blocked senders waiting for matching receive operations. Each element on the channel is a value that is being sent and a reference to the blocked parasite. There are no blocked receivers on this channel. Part 1 of the figure shows the parasite S_1 invoking a send on this channel with a value `v`. Since there are no blocked receivers, we add the value `v` and the reference to S_1 on the channel, as shown in part 2 of the figure. Host T_1's stack top is updated to point to the next parasite in the linked list of parasites as maintained by the host thread. Here, the next parasite is the parasite at the bottom, S_3. T_1 can continue to run other parasites on the stack as long as suspended parasites are available. The host thread blocks when all its parasites are blocked. S_1 is unblocked when it pairs up with a matching receive operation. Fig. 8 shows parasite S_4 receiving a value on the channel C_1. S_4 unblocks S_1 and receives the value v sent by S_1.

For a receive operation with no senders already available on the channel, the parasite enqueues a reference to itself and a reference to the as-of-yet unknown value. A matching sender parasite changes the state of the blocked parasite to suspended from blocked, and stores the value of the communication on the provided reference. When the parasite resumes, it uses the values of the reference as the argument to its continuation.

Notably, unblocking a blocked parasite on a matching communication does not involve locking the target thread and can concurrently be performed without affecting the execution of target thread. If the parasites were extended to support CML style selective communication, an atomic memory operation would be necessary and sufficient to validate one of the communication choices and invalidate others.

3.7 Interaction with GC

Just like host threads in the language runtime, the garbage collector must be aware of parasitic threads and the objects it references. But since parasites always exist as a part of the host thread's stack, the garbage collector can treat parasites as regular frames; the collector is aware of the active portion of the parasite's context (that information is associated with each frame), and the existence of holes in the stack, both of which can be skipped during collection.

3.8 Specialization of Communication

Our semantics expressed a useful optimization that allowed a direct hand-off of values between communicating parasites located in the same thread. We leverage Reppy's compiler analysis for specialization of channels to discover write-once channels (19). A typical programming idiom of CML is to create a local channel on which a result is propagated between an asynchronous computation to its parent. Since parasites never migrate, such communications can be optimized to a local value exchange.

4. Results

To evaluate the performance of our raw message passing primitives, we consider micro-benchmarks that test parasite performance against CML's threads. In addition to these micro benchmarks, we evaluate parasitic threads on some well-known communication intensive benchmarks: `Barnes-hut` N-body simulation, `Mandelbrot` set fractal generation and `Raytrace`. The parallel versions of these programs were encoded in a master-slave model, where a master thread partitions the work into equal sized chunks and distributes the work among n workers, where n is proportional to the number of available processors. The master communicates asynchronously with workers since there is no *a priori* order in which workers initiate requests for new work from the master.

4.1 Evaluation

The benchmarks were run on 16-way AMD Opteron 865 server with 8 processors, each containing two symmetric cores, and 32 GB of total memory, with each CPU having its own local memory of 4 GB. Access to non-local memory is mediated by a hyper-transport layer that coordinates memory requests between processors. MLton uses 64-bit pointer arithmetic, SSE2 instructions for floating point calculations, and is IEEE floating point compliant.

4.1.1 Micro-benchmarks

We consider three micro-benchmarks to quantify parasite performance (see Fig. 9). The first `Non-blocking` creates n threads, each of whom write a value to a channel. We then consider two different implementations to read these values: (1) creating n CML threads each of whom read a single element; and (2) creating n parasites on a single host thread, each of whom reads a single element. Parasites significantly outperform the threading implementation, up to 13X speedup when $n = 10K$.

Our second benchmark, `Blocking`, is similar to the first except that receivers execute before senders. This benchmark measures the cost of rescheduling blocked parasites against the cost of scheduling blocked threads. Our results are similar to the non-blocking case.

The first two benchmarks measures the impact of a parasite implementation on channel contention. Our last benchmark `N-way` quantifies contention on thread queues. We create n senders each of whom writes to a separate channel. We then perform n receive

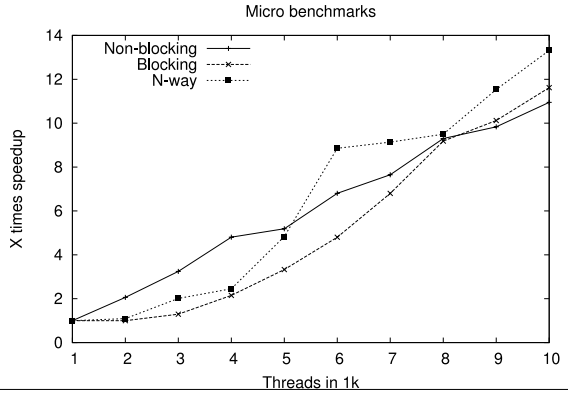

Figure 9. Speedup for parasitic implementation of micro benchmarks compared to corresponding CML thread implementations on 16 cores.

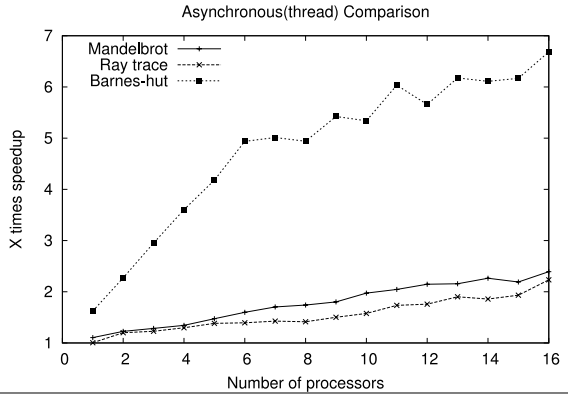

Figure 10. Speedup for Mandelbrot, Raytrace and Barnes-hut benchmarks implemented using parasites compared to a corresponding asynchronous implementation using CML threads.

operations on these *n* distinct channels. Lightweight threading creates *n* threads, and the parasite version creates *n* parasites; there are no costs due to blocking. As before, we see that scheduling overheads of *n* threads has a notable impact on scalability compared to using parasites.

4.1.2 Parallel Benchmarks

Compared to single core performance, `Mandelbrot` exhibits 14.2X speedup, `Raytrace` shows 14.7X speedup, and `Barnes-Hut` demonstrates 8X [1] speedup when run on 16 cores using parasites. We choose inputs for our benchmarks so they ran on the order of several minutes. Fig. 10 shows parasite performance compared to an asynchronous version (running on the same number of cores) using lightweight CML threads.

We also present salient profile data for parasite and asynchronous versions of the benchmarks on 16 core runs in Table 1. `Spawns` represent the number of host threads and parasites created in the parasitic version and the number of threads spawned in the asynchronous CML version, resp. The `Avg. size` column shows the average stack size of parasites, and the average stack size of threads in the asynchronous CML version, resp. `Comm` is the number of communication actions performed within an asynchronous computation, `%Blocked` represents the fraction of parasites that block on a synchronous communication at least once, and `%Preempted` is a measure of the number of times a parasite is preempted on average. For example, a `%Preempted` value of 200 indicates that every parasite is preempted twice on average.

The statistics reveal that parasites consume substantially less memory than CML's lightweight threads, often requiring less than 100 bytes to execute. This occurs because parasites are mostly engaged in communication actions, and often block; allowing them to use a host's resources (essentially, acting as a passive agent when blocked) helps reduce overall thread management and scheduling overheads. The preemption column indicates that parasites are not always short-lived, and often perform enough computation to trigger a context-switch, indicating that a cooperative scheduling policy may be ineffective or non-trivial to implement efficiently, requiring yield points to be well-placed.

Our benchmark results show a general decay in performance of the asynchronous (non-parasite) implementation in comparison to an implementation based on parasites. This decay is a result of the overheads incurred due to CML thread creation, management and context switches. Although CML threads are extremely lightweight, on the order of 500 bytes, scheduling and synchronization exact a toll on performance. Parasitic threads, are on average an order of magnitude smaller than CML threads, are often zero-copy schedulable, and require significantly less underlying synchronization.

A thread blocked on a channel, needs to be placed on a processor queue when it is unblocked. Unfortunately, this requires synchronization on the processor queue the thread wishes to be enqueued upon. Hence, for each pair of communications, an atomic action (implemented via a low-level processor lock operation) is required on the processor queue [2]. The contention on the processor queue increases significantly with an increase in the number of communicating threads performing concurrent communication, even though such threads may be communicating over distinct channels. These overheads quickly erode the gains obtained by on-demand work distribution via asynchronous computation. In contrast, a parasite performing a matching communication directly unblocks the blocked parasite, without any need for locking. It is this feature that contributes most notably to its improved relative performance.

4.1.3 Swerve Case Study

`Swerve` consists of a collection of modules that communicate using CML's message-passing operations. There are four critical modules of interest: (a) a *listener* that processes incoming requests; (b) a *file processor* that handles access to the underlying file system; (c) a *network processor* that communicates with the client and, (d) a *timeout* manager that regulates the amount of time allocated to serve a request. There is a dedicated listener for every distinct client, and each request received by a listener is delegated to a server thread responsible for managing that request. Requested files are broken into chunks and packaged as messages sent back to the client.

Our case study focused on the file and network processors. A new thread is spawned for each client request. This thread subsequently spawns one thread each for file server and network processor. File processor traverses the local directory to get the file, splits the file into chunks and sends it to the network processor. Network processor gets the file chunk and sends it over the socket to the client. These components also take care of handling connection termination errors, file unavailability, etc. They also synchronize with the timeout manager to poll for timeout signals.

[1] Barnes-Hut requires synchronization between slave threads and the master at every iteration of the benchmark.

[2] The use of lock-free queues alleviates this slightly, but once contention increases performance decreases correspondingly.

Benchmark	LOC	Spawns	Avg size (bytes) Parasites	Threads	Comm	% Blocked	% Preempted
Mandelbrot	151	4122	40	892	4115	99	90
Ray trace	2512	619	44	1284	618	94	43
Barnes-hut	1251	16016	43	1184	15875	75	24
Swerve	23671	21747	208	3104	72231	82	38

Table 1. Benchmark statistics on 16 core runs.

An important aspect of the implementation is that the file processor and the network processor proceed in lock step. The file processor reads a chunk of the file and performs a synchronous send to the network processor. Only after the network processor has received the chunk can the next chunk be processed. Typically, network communication is slower than disk I/O. Hence, file processing often blocks waiting for network processors to signal readiness to receive the chunk.

We make use of parasites to achieve asynchrony in this scenario. A wrapper function converts I/O actions to synchronous communication. This ensure that the file read is preemptible and does not block the entire thread. The file processor asynchronously processes reading the file with respect to the communication it undertakes with the network processor.

The file is read in 8KB chunks. Each chunk read is performed asynchronously. Since the chunks are being read from a stream, the parasites must read the files in order, and send it to the network processor in the same order. This synchronization between the parasites is implemented using our synchronous communication primitives.

```
1  fun chunkProcessor (prevFileCh, nextFileCh,
2                      prevNWch, nextNWch)
3    let
4      val _ = recv (prevFileCh)
5      val chunk = read (fileStrm, chunkSize)
6    in
7       send (nextFileCh, ());
8       recv (prevNWCh);
9       send (nwProcCh, chunk);
10      send (nextNWCh, ())
11   end
```

The file processor creates one parasitic thread for every file chunk that evaluates the procedure shown above. Since file chunk reads have to happen in order, the parasitic thread waits for the previous parasitic thread to finish reading this chunk (line 4). After it is unblocked, it reads the chunk (line 5), and finally signals the next waiting parasite that it can continue (line 6). This ensures an ordering in the chunk reads. The same synchronization mechanism is used for sending the chunk to the network processor (line 8). Performing chunk reads asynchronously in this way allows overlap of file reads with network communication, reducing the bottleneck in the original implementation.

We tested the system with file sizes from a few kilobytes to 20 MB, and compared the parasite implementation with the original version of Swerve executing on 16 cores using only synchronous communication. With our changes incorporated into the system, we see a 25% overall raw speed up in the server when using 16 cores. Additionally, the throughput of the file processor increased 75% due to the reduction in the time each file descriptor was kept open. The statistics presented in Table 1 correspond to the server processing 10 requests of a file size of 16 MB.

We then compared our system against a Swerve implementation where all parasites were replaced by CML threads, effectively implementing asynchrony through lightweight threading; the results are shown in Fig. 11. The parasitic version remains considerably faster than the asynchronous version and speed up increases as the file size is increased. More significantly, lightweight threading actually results in *poorer* performance compared to the synchronous version, as described above. Since the chunk size is 8KB, the asynchronous threading version creates 2048 threads to read a 16 MB file, each of size 3 KB on average (See Table 1); in contrast, the memory overhead associated with each parasite is just 208 bytes. The cost of synchronization on thread and channel queues among these asynchronous threads becomes substantial, and leads to notable degradation as file size increases. These overheads outweigh any benefits that accrue from asynchronous evaluation. On the other hand, the parasite implementation profitably exploits asynchrony without thread management and scheduling overheads.

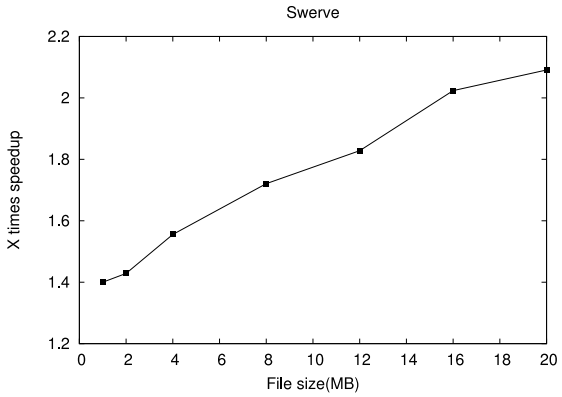

Figure 11. Speedup for Swerve compared to a corresponding asynchronous implementation using lightweight threads on 16 cores.

5. Related Work

There are a number of languages and libraries which support varying kinds of message-passing styles. Systems such as MPI (13; 25) support per-processor static buffers, while CML (21), Erlang (3), F# (24), and MPJ (4) allow for dynamic channel creation. Although MPI supports two distinct styles of communication, both asynchronous and synchronous, not all languages provide primitive support for both. For instance, Erlang's fundamental message passing primitives are asynchronous, while CML's primitives are synchronous. We believe the benefits of parasitic threads can be leveraged regardless of the underlying fundamental message passing primitive used.

There has been much interest in lightweight threading models ranging from using runtime managed thread pools such as those found in C# (22) and the Java util.concurrent package (12), to continuation-based systems found in functional languages such as Scheme (17; 11), CML (21), Haskell (8), and Erlang (3). Parasitic

threads can be viewed as a form of lightweight threading, though there are a number of key differences. First, from the perspective of a host thread, initiating a parasitic thread is akin to making a procedure call; the parasite executes using the same resources as the host, and has its computation initially scheduled before the host as well. Second, compiler analysis is critical to enable parasitic threads to be managed with zero-copy overheads when possible, even when they run in the middle of a host thread's stack. Third, because parasitic threads are attached to a host, they incur no locking overheads for scheduling.

The parasitic threads described here are similiar in some respects to the lightweight threads found in Capriccio (26). Threads in Capriccio are comprised of small, non-contiguous stacks that can grow or shrink at runtime. Like our approach, Capriccio relies on compiler support to limit the amount of stack space that is preallocated to a thread. While a non-contiguous stack implementation is feasible even for parasitic threads, it would have entailed substantial re-engineering of the MLton compiler and runtime; fortunately, as our benchmark results indicate, most parasitic threads consume a very small amount of stack space, often less than hundred bytes. More significantly, unlike parasitic threads, Capriccio does not support specialized message passing primitives. In Multi-Mlton, communication operators are aware of parasites, and can unblock waiting parasites quickly, with no synchronization overheads.

Kilim (23), is an actor-based (1) model implemented in Java using lightweight threading. Kilim assumes a programming environment that utilizes no shared memory and no locks. In comparison, parasitic threads impose no constraints on the code they execute. Additionally, Kilim requires copying of state associated with each lightweight fiber on a context switch. Parasitic threads are designed to incur no copy overheads on context switches whenever possible. Fibers and the actors associated with Kilim actors are cooperatively scheduled and thus impact fairness guarantees.

Parasites also share some similarity to dataflow (10; 18) languages, insofar as they represent asynchronous computations that block on dataflow constraints; in our case, these constraints are manifest via synchronous message-passing operations. Unlike classic dataflow systems, however, parasites are not structured as nodes in a dependency graph, and maintain no implicit dependency relationship with the thread that created it. CML (21) supports buffered channels with asynchronous sends. While parasites can be used to encode asynchronous sends, unlike buffered channels, they provide a general solution to encode arbitrary asynchronous computation.

Work stealing (5; 2) is a well-known technique for load balancing multithreaded tree-structured computations, and has been used effectively in languages like Cilk (6) to improve performance. Intuitively, work stealing parasitic threads would appear to be beneficial since a given host may have many suspended parasites ready to execute, but which cannot. However, allowing parasites to be stolen by other threads would substantially complicate the overall system design. This is partly because, (a) unlike a typical work-stealing scheme, parasites are associated with an execution context, and (b) additional synchronization among threads would be required to migrate a parasite from one host to another.

6. Conclusions

This paper describes a novel threading mechanism called parasites that enables lightweight asynchronous computation. The key distinguishing feature of parasites is its ability to execute using the resources of a host thread. This mechanism enables higher-level programmability without imposing additional scheduling and thread management overheads.

References

[1] Gul Agha. An Overview of Actor Languages. *SIGPLAN Not.*, 21(10):58–67, 1986.

[2] Kunal Agrawal, Yuxiong He, and Charles E. Leiserson. Adaptive Work Stealing with Parallelism Feedback. In *PPoPP*, pages 112–120, 2007.

[3] Joe Armstrong, Robert Virding, Claes Wikstrom, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 2nd edition, 1996.

[4] Mark Baker and Bryan Carpenter. Mpj: A proposed java message passing api and environment for high performance computing. In *IPDPS*, pages 552–559, 2000.

[5] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *J. ACM*, 46(5):720–748, 1999.

[6] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*, pages 212–223, 1998.

[7] Glasgow Haskell Compiler. `http://www.haskell.org/ghc`.

[8] Tim Harris, Simon Marlow, , and Simon Peyton Jones. Haskell on a Shared-Memory Multiprocessor. In *Haskell Workshop*, pages 49–61, 2005.

[9] Intel. http://software.intel.com/en-us/articles/intel-mpi-benchmarks.

[10] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in Dataflow Programming Languages. *ACM Comput. Surv.*, 36(1):1–34, 2004.

[11] D. A. Kranz, R. H. Halstead, Jr., and E. Mohr. Mul-T: A High-Performance Parallel Lisp. In *PLDI*, pages 81–90, 1989.

[12] Doug Lea. *Concurrent Programming in Java(TM): Design Principles and Pattern*. Prentice-Hall, 2nd edition, 1999.

[13] Guodong Li, Michael Delisi, Ganesh Gopalakrishnan, and Robert M. Kirby. Formal Specification of the MPI-2.0 Standard in TLA+. In *PPoPP*, pages 283–284, 2008.

[14] M. Matthias Felleisen and Dan Friedman. Control operators, the SECD Machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, pages 193–217, 1986.

[15] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[16] MLton. http://www.mlton.org.

[17] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. In *LFP*, pages 185–197, 1990.

[18] Rishiyur Nikhil and Arvind. *Implicit Parallel Programming in pH*. Morgan-Kaufmann, 2001.

[19] John Reppy and Yingqi Xiao. Specialization of CML Message-Passing Primitives. In *POPL*, pages 315–326, 2007.

[20] John Reppy and Yingqi Xiao. Towards a Parallel Implementation of Concurrent ML. In *DAMP*, January 2008.

[21] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[22] C# Language Specification. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf.

[23] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-Typed Actors for Java. In *ECOOP*, pages 104–128, 2008.

[24] Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F#*. Apress, 2007.

[25] Hong Tang and Tao Yang. Optimizing Threaded MPI Execution on SMP Clusters. In *ICS*, pages 381–392, 2001.

[26] Rob von Behren, Jeremy Condit, Feng Zhou, George Necula, and Eric Brewer. Capriccio: Scalable Threads for Internet Services. In *SOSP*, pages 268–281, 2003.